

# 1. 本周工作

## 1.1 阅读《C++ concurrency in action》Chapter 2. Managing threads

这本书主要是对C++11标准(ISO标准)颁布的线程库<thread>，及多线程编程进行了比较详尽的介绍与解读。

选择这本书的原因，是由于当下C\C++对底层硬件的访问能力，使其在网络编程中得以广泛应用。学习遵循标准的多线程编程技巧，有助于实现高清拼接屏项目。

### 1.1.1 Basic Thread Management

#### 1.1.1.1 Launching a thread

通过一个可调用的对象创建一个线程对象，这个对象通过**拷贝**的方式传递到线程对象的构造函数中。可以通过**detach**方式启动线程：

```
struct func
{
    int & _i;

    func(int& i): _i(i) {}

    void operator() ()
    {
        // to sth.
    }
};

void oops()
{
    int local = 0;
    func func_obj(local);
    thread t(func_obj);
    t.detach();
    cout << "oops done" << endl;
}
```

注意到oops不会等线程t的任务结束才退出。

亦即在t线程任务结束前，栈地址空间可能会被收回，这将导致func\_obj中存有一份对已被回收的局部变量的引用，对该变量的操作是未定义的(undefined)。

#### 1.1.1.2 Waiting for a thread to complete

与detach不同，t.join()将阻塞oops()的执行。

亦即在t任务结束后，oops()中的"oops done"才会被输出。

join操作是简单粗暴的——等待或不等待线程结束，在第四章中，介绍了用**条件变量**或**future**来进行不同粒度的控制等待的方式：

- 检查线程是否完成
- 限时等待

#### 1.1.1.3 Waiting in exceptional circumstances

本节阐述了在函数异常退出，导致栈空间被回收的情况。

```

void f()
{
    thread t(func);
    try
    {
        do_something_in_current_thread();
    }
    catch(...)
    {
        t.join();
    }
    t.join();
}

```

如果没有这个try-catch块，函数可能由于发生异常，意外退出，进而导致一个与系统线程相关联的线程对象(thread object), 这会导致整个程序意外终止(abort)。

一种做法如上所示，用两个t.join()维护。但随着代码规模的增长，这种做法不仅写法丑陋，而且易出错。遂提出了thread\_guard, 用以保证在函数被意外终止的情况下，仍可以启动已创建的线程对象：

```

class thread_guard
{
public:
    explicit thread_guard(thread& t): _t(t) {}
    ~thread_guard()
    {
        if (t.joinable())
        {
            t.join();
        }
    }
    thread_guard(const thread_guard&) = delete;
    thread_guard& opeartor=(thread_guard const&) = delete;

private:
    thread _t;
};

```

由于thread\_guard作为局部变量会在函数结束时被析构，在析构函数中调用t.join()方法，可以保证即使在函数意外终止的情况下，线程仍被启动。

#### 1.1.1.4 Running threads in the background

举例描述了t.detach()方法，是用于将线程放在后台运行；

而非像t.join()那样，令当前线程等待线程t结束后才继续执行。

#### 1.1.2 Passing arguments to a thread function

默认按值传递(by copied)

```

thread t(func, arg1, arg2);

```

通过引用传递：使用std::ref

```

thread t(func, arg1, std::ref(arg2));

```

调用成员函数：传递类的成员函数指针及类的对象指针

```
class X
{
public:
    void do_lengthy_work();
};

X x;
thread t(&X::do_lengthy_work, &x);
```

通过std::move处理只允许被moved不允许被copied的对象(movable but uncopyable)

```
void f(unique_ptr<big_obj>);

unique_ptr<big_obj> p(new big_obj);
thread t(x, move(p));
```

### 1.1.3 Transferring ownership of a thread

运用move转移thread的拥有权。

### 1.1.4 Choosing the number of threads at runtime

std::hardware\_concurrency()返回在该程序下可并行的线程数，在多核系统中，该数目为CPU的数量。当返回0时，意味着无法获得该数量。

受CPU核数限制，过多增加线程数目无法增加并行的线程数，同时将CPU时间浪费在无谓的线程切换上。

### 1.1.5 Identifying threads

线程标识符的类型为std::thread::id

std::this\_thread::get\_id()将返回当前正在执行的线程的id。

## 1.2 多线程编程实践

基于书本内容，对下述情形分别进行编程实践：

- detach线程
- join线程
- 未处理的线程
- thread\_guard, 线程守护

```
→ chapter2 1a
total 256
-rwxr-xr-x 1 stephen staff 16K Oct 12 14:40 detach
-rw-r--r-- 1 stephen staff 204B Oct 12 15:17 detach.cpp
-rwxr-xr-x 1 stephen staff 16K Oct 12 14:38 join
-rw-r--r-- 1 stephen staff 202B Oct 18 21:45 join.cpp
-rw-r--r-- 1 stephen staff 132B Oct 15 10:35 non_management_thread.cpp
-rwxr-xr-x 1 stephen staff 16K Oct 12 16:50 run
-rw-r--r-- 1 stephen staff 144B Oct 12 16:50 run.cpp
-rwxr-xr-x 1 stephen staff 16K Oct 15 10:36 test
-rwxr-xr-x 1 stephen staff 21K Oct 12 15:06 thread_guard
-rw-r--r-- 1 stephen staff 164B Oct 12 15:16 thread_guard.cpp
-rw-r--r-- 1 stephen staff 311B Oct 19 00:21 thread_guard.h
```

## 2. 下周工作

阅读Chapter 3. Sharing data between threads;

阅读姜老师提供的材料，了解一些新的主题；  
参加CNCC.